SEHH2239 Data Structures
Lecture 11

Learning Objectives:

- To describe the purpose of hashing
- To explain the collisions in hashing
- To solve collisions with linear probing and separate chaining

Introduction

- Consider the problem of **searching** an array for a given value
  - If the array is not sorted,
    - If the value isn't there, we need to search all n elements
    - If the value is there, we search n/2 elements on average
  - If the array is sorted, we can do a binary search
    - About equally fast whether the element is found or not
  - It doesn't seem like we could do much better
    - That is **constant time search**?
    - We can do it if the array is organized in a particular way

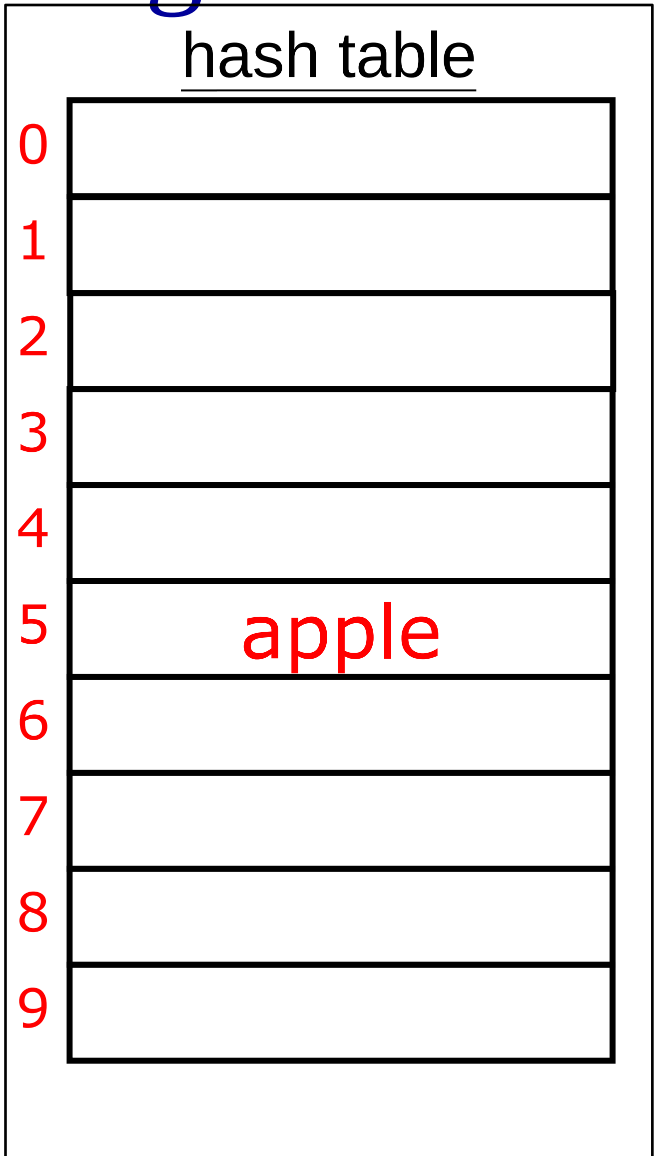
Hashing

- Recall the {key, value} pairs.
- Questions:
  - How to store the keys?
- **Hashing** is a technique that determines the ***index*** of a ***key***.
  - Not search for the item
- **Hash Table**
  - Array storing values indexed by a hash function

Example hashing

- Suppose our hash function gave us the following values:

    `hashCode("apple") = 5`



Example hashing

- Suppose our hash function gave us the following values:

  hashCode("apple") = 5
  hashCode("watermelon") = 3
  hashCode("grapes") = 8

Example hashing

- Suppose our hash function gave us the following values:

  hashCode("apple") = 5
  hashCode("watermelon") = 3
  hashCode("grapes") = 8
  hashCode("cantaloupe") = 7
  hashCode("kiwi") = 0
  hashCode("strawberry") = 9
  hashCode("mango") = 6
  hashCode("banana") = 2

hash table

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Getting value from key

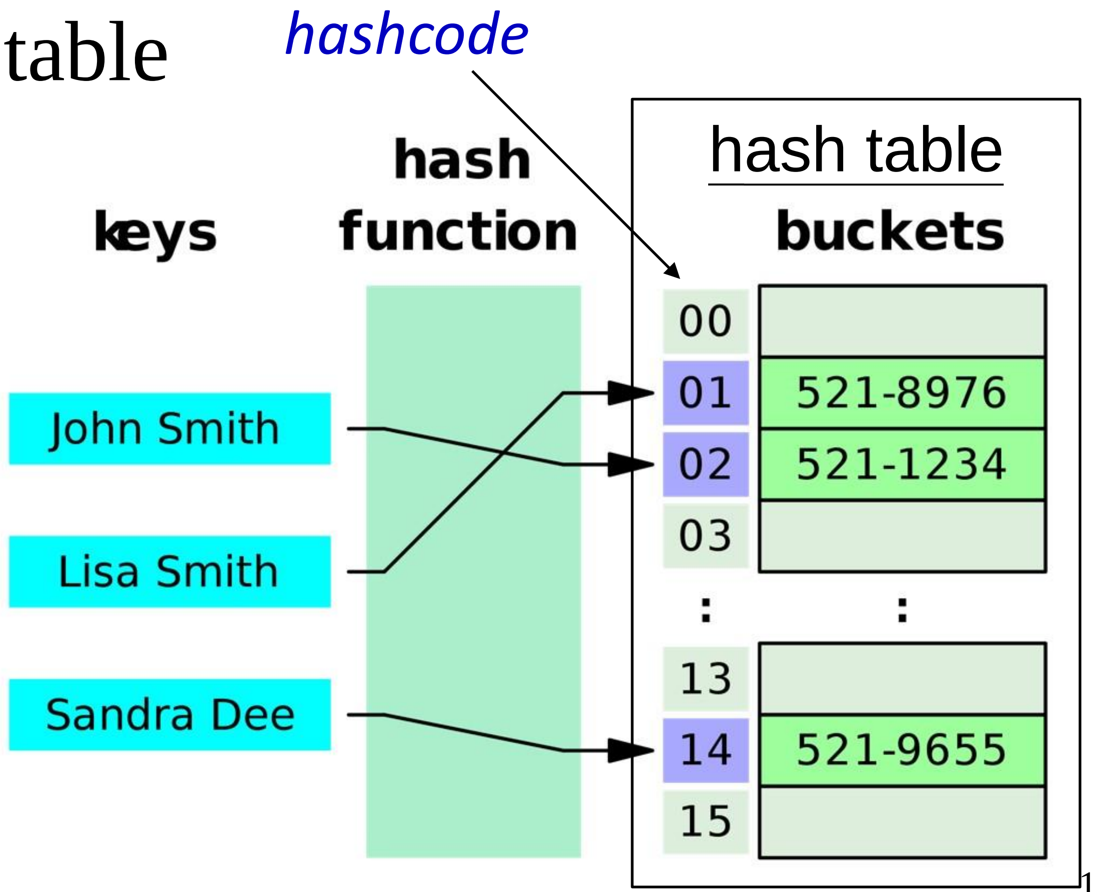
- Storing {key, value} pair
  - We use a ***key*** to find a place in the hash table
  - The associated ***value*** is the information we are trying to look up
- Example:
  - *hash*(robin) = 142
  - *hash*(owl) = 148

	key	*value*
. . .		
141		
142	robin	robin info
143	sparrow	sparrow info
144	hawk	hawk info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info

Hash function

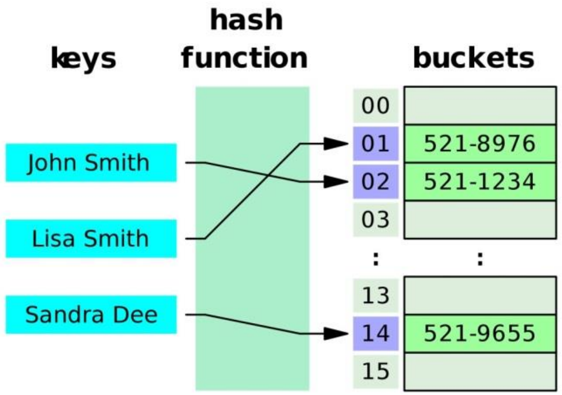
Hash function

- A hash function takes a search *key* and produces the *integer index* of an element in the hash table



Use of Hash function

- Uses a 1D array (or table) table[0:b-1].
  - Each position of this array is a **bucket**.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function ***f*** that converts each key **k** into an index in the range **[0, b-1]**.
  - ***f(k)*** is the **home bucket** for key **k**.
- Every dictionary pair (key, element) is stored in its home bucket table[*f*(key)].



Looking up with Hash function

- Given a key to look up for, it would tell us exactly where in the array to look
  - If it's in that location, it's in the array
  - If it's not in that location, it's not in the array

Hash function

- A hash function is a function that:
  - When applied to an Object, returns a number (index)
  - When applied to *equal* Objects, returns the *same* number for each
  - When applied to *unequal* Objects, is *very unlikely* to return the same number for each
- Preliminary examples of hash functions:
  - hash(X) = X/n where n is 11
  - hash(X) = X % noOfBuckets

Example of Hashing

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is *h(key)= key/11*.
- Pairs are stored in table as below:

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Perfect Hashing and Collision

Prefect Hashing

- A Prefect hashing maps each *search key* into a ***different integer*** that is suitable as an index to hash table
- Efficiency
  - Can result in O(1) search times

Examples of Perfect Hashing

hash table

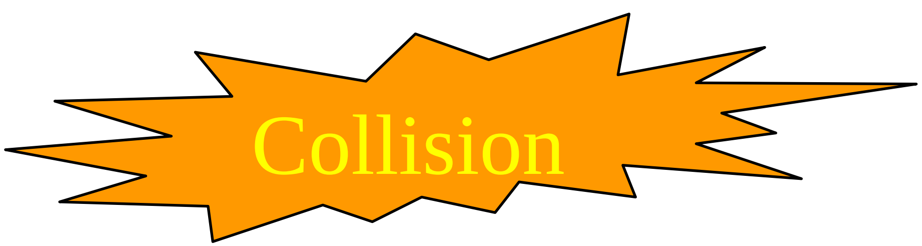
0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

What Can Go Wrong?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

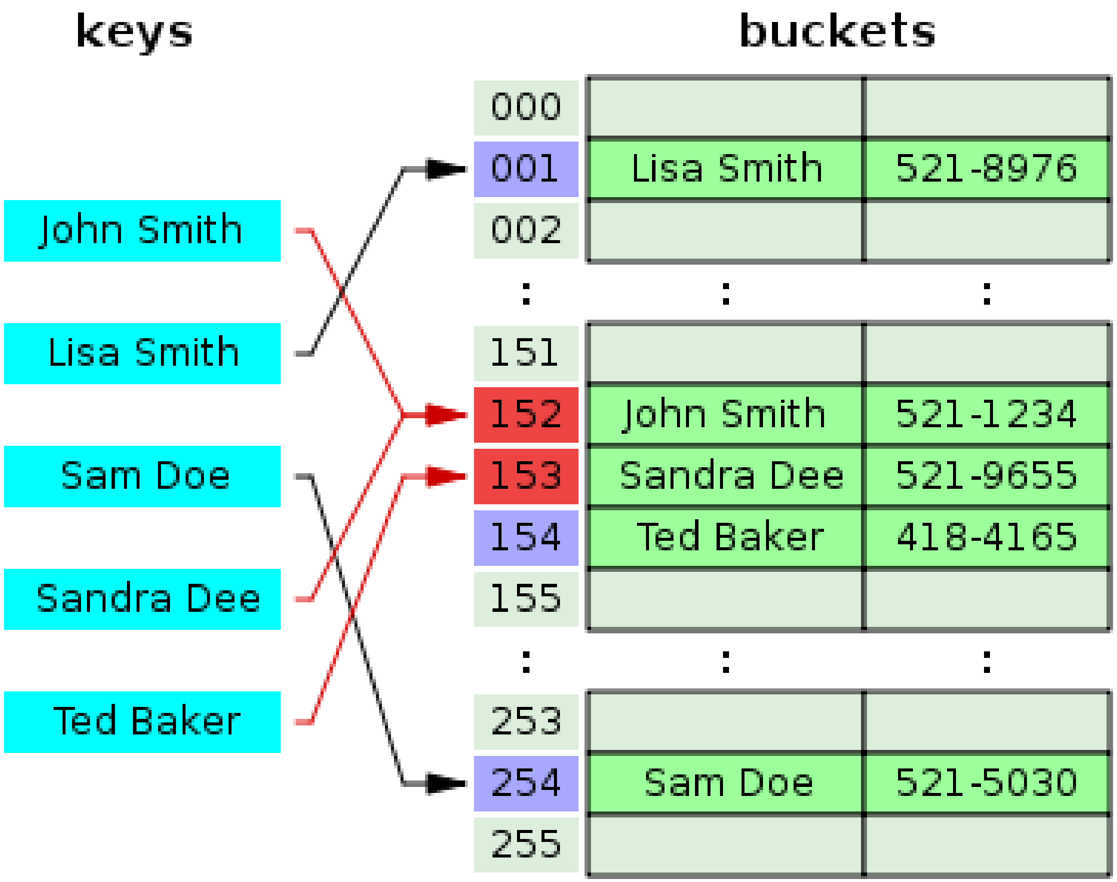
- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The home bucket for (26,g) is already occupied.



Collisions

- When two values hash to the same array location, this is called a collision
- Collisions are normally treated as “**first come, first served**”—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location

Collision


keys
buckets
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
:
151
152
153
154
155
:
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
  - **Solution #1:** Linear probing - search from there for an **empty location**
    - Can stop searching when we find the value *or* an empty location
    - Search must be end-around
  - **Solution #2:** Use the array location as the header of a **linked list** of values that hash to this location
- All these solutions work, provided:
  - We use the same technique to *add* things to the array as we use to *search* for things in the array

Collision Handling by Linear Probing

Linear probing

- all entry records are stored in the array itself.
- When a new entry has to be inserted, compute the hashed value
  - If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index (as the Prefect hashing).
  - If the slot is occupied, it proceeds in some probe sequence until it finds an unoccupied slot.

Insertion - Case 1

- Suppose you want to add seagull to this hash table
- Also suppose:
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is empty
- Therefore, put seagull at location 145

Searching - Case 1

- Suppose you want to look up seagull in this hash table
- Also suppose:
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is not empty
  - table[145] == seagull !
- We found seagull at location 145

Searching - Case 2

- Suppose you want to look up `cow` in this hash table
- Also suppose:
  - `hashCode(cow) = 144`
  - `table[144]` is not empty
  - `table[144] != cow`
  - `table[145]` is not empty
  - `table[145] != cow`
  - `table[146]` is empty
- If `cow` were in the table, we should have found it by now
- Therefore, it isn't here

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Case 3

- Suppose you want to add hawk to this hash table
- Also suppose
  - hashCode(hawk) = 143
  - table[143] is not empty
  - table[143] != hawk
  - table[144] is not empty
  - table[144] == hawk
- hawk is already in the table, so do nothing in adding.

Case 4

- Suppose:
  - You want to add cardinal to this hash table
  - hashCode(cardinal) = 147
  - The last location is 148
  - 147 and 148 are occupied
- Solution:
  - Treat the table as **circular**; after 148 comes 0
  - Hence, cardinal goes in location 0 (or 1, or 2, or ...)

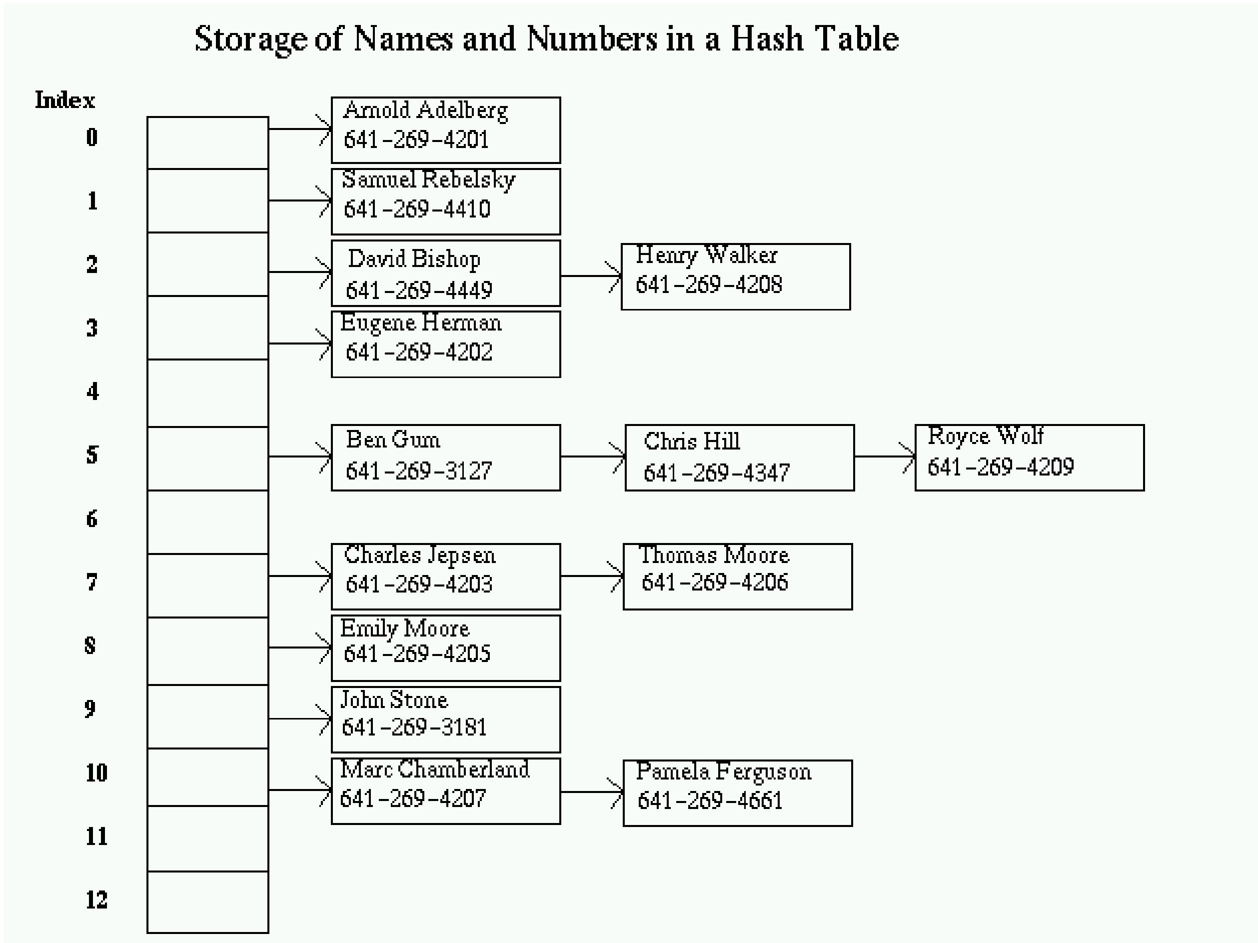
. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl

Collision Handling by Separate chaining

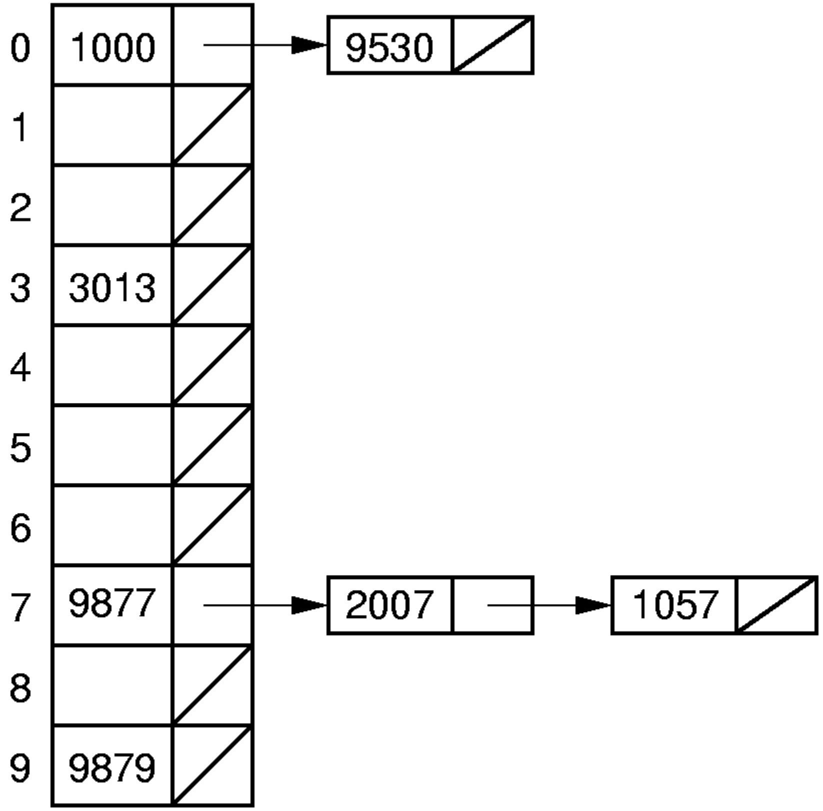
Separate chaining

- In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list.
- If there is any collision, store both the elements in the same linked list.

Separate chaining – I



Separate chaining - II



Overflow

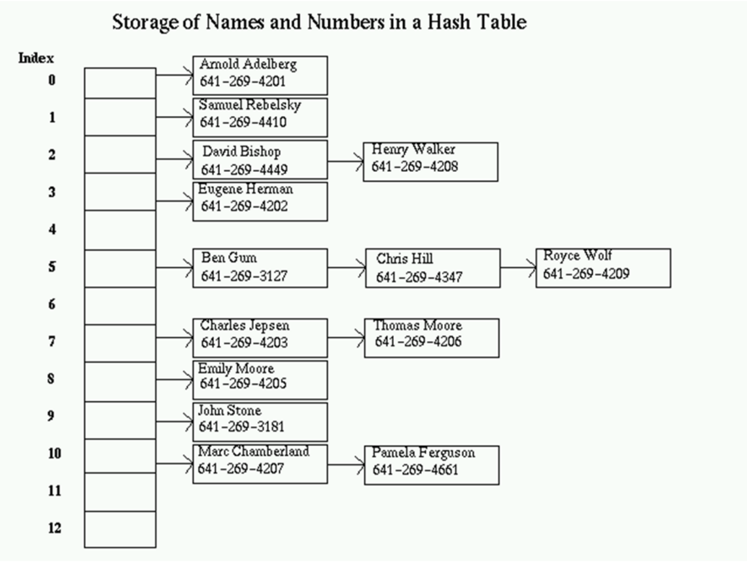
Overflow

(3,d)	(11,m)	(22,a)	(33,c)	(45,k)	(55,o)	(73,e)	(85,f)

- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.

Handling Overflow

- Linked list can also resolve the problem, as linked list can be expanded to increase the capacity.



Hashing in Python and Efficiency

Hash table

- In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.
  - The keys of the dictionary are hashable i.e. the are generated by hashing function which generates unique result for each unique value supplied to the hash function.
  - The order of data elements in a dictionary is not fixed.

Example

```python
# Declare a dictionary
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

# Accessing the dictionary with its key
print("dict['Name']: ", dict['Name'])
print("dict['Age']: ", dict['Age'])
```

```
Output:
dict['Name']: Zara dict['Age']: 7
```

Prefect Hashing

- Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time.

Good Hash Functions

- General characteristics of hash functions *h*(*key*). Any function can be a hash function that **distributes entries uniformly** throughout the hash table.
- Good hash functions:
  - Minimize collision
  - Be fast to compute
- There are many ways to implement hash functions, e.g. to use *prime number division*, *mid square*, *move or folding* just to mention a few, but they are beyond the scope of the class.

Efficiency

- Hash tables are actually surprisingly efficient
- Until the table is about 70% full, the number of probes (places looked at in the table) is typically only 2 or 3
- Sophisticated mathematical analysis is required to *prove* that the expected cost of inserting into a hash table, or looking something up in the hash table, is constant time
- Even if the table is nearly full (leading to occasional long searches), efficiency is usually still quite high

Summary of key terms

- Hash
  - hashing
  - Hash function
  - Prefect Hashing
- Collision
  - Linear probing
  - Separate chaining
- Overflow